Fast Finite Difference Poisson Solvers on Heterogeneous Architectures

Pedro Valero-Lara[a,c], Alfredo Pinelli[b], Manuel Prieto-Matias[c]

[a] *Unidad de Modelización y Simulación Numérica, CIEMAT, Madrid, Spain. pedro.valero@ciemat.es*
[b] *School of Engineering and Mathematical Sciences, City University London, U.K.. Alfredo.Pinelli.1@city.ac.uk*
[c] *Departamento de Arquitectura de Computadores, Facultad de Informática, Universidad Complutense de Madrid (UCM), Spain. mpmatias@dacya.ucm.es*

Abstract

In this paper we propose and evaluate a set of new strategies for the solution of three dimensional separable elliptic problems on CPU-GPU platforms. The numerical solution of the system of linear equations arising when discretizing those operators often represents the most time consuming part of larger simulation codes tackling a variety of physical situations. Incompressible fluid flows, electromagnetic problems, heat transfer and solid mechanic simulations are just a few examples of application areas that require efficient solution strategies for this class of problems. GPU computing has emerged as an attractive alternative to conventional CPUs for many scientific applications. High speedups over CPU implementations have been reported and this trend is expected to continue in the future with improved programming support and tighter CPU-GPU integration. These speedups by no means imply that CPU performance is no longer critical. The conventional CPU-control-GPU-compute pattern used in many applications wastes much of CPU's computational power. Our proposed parallel implementation of a classical cyclic reduction algorithm to tackle the large linear systems arising from the discretized form of the elliptic problem at hand, schedules computing on both the GPU and the CPUs in a cooperative way. The experimental result demonstrates the effectiveness of this approach.

Keywords: Fast Finite Difference Poisson Solvers, Parallel Computing, CPU-GPU Heterogeneous Architectures.

1. Introduction

The era of single-threaded processors has come to an end due to the limitation of the CMOS technology and in response, most hardware manufactures are designing and developing multi-core processors and specialized hardware accelerators such as GPUs [6, 16, 17]. As a consequence, applications can only improve their performance if they are able to exploit the available parallelism of the new architectures.

In this paper we study the implementation of a fast solver based on a block cyclic reduction algorithm to tackle the linear systems that arise when discretizing a three dimensional separable elliptic problem with standard finite difference. A clear example of the importance of dealing efficiently with three dimensional elliptic systems is found in the numerical simulation of incompressible fluid flows. Indeed, the most time consuming part of almost any incompressible unsteady Navier Stokes solver (i.e., incompressible fluid dynamic simulation codes) is related to the solution of a *pressure Poisson* equation at each time step (see for instance [12]). The achievement of a satisfactory computational efficiency to tackle this class of elliptic partial differential equations is therefore a key issue when simulating unsteady fluid flow processes (turbulent flows for instance).

Other authors have addressed topics which are somehow related to the present contribution. [5] analyzes the performance of a block tridiagonal benchmark on GPUs. This is the first known implementation of a block tridiagonal solver in CUDA but the pattern of the block matrices they analyzed differ from our target problem. The sub-matrix element rank (m) was assumed to be small ($m = 5$). In our case both m and the arithmetic intensity of problem are higher.

For distributed multicore clusters, the BCYCLIC algorithm developed by Hirshman et al. [3] is able to solve linear problems with dense tridiagonal blocks. Our target algorithm, the BLKTRI code [13] is not well-suited for dense blocks but it is the most popular approach for solving block tridiagonal matrices which arise from separable elliptic partial differential equations.

Many authors have studied the implementation of scalar tridiagonal solver on GPUs [8, 7, 1, 2, 4]. D. Göddeke et al. [8] proposed an efficient implementation of the Cyclic Reduction (CR) algorithm, which is used as a line smoother in a multigrid solver. Yao Zhang et al. [7] proposed some hybrid algorithms that combine CR with other tridiagonal solvers such as Parallel Cyclic Reduction (PCR) or Recursive Doubling (RD). More recently, H. Kim

et al. [4] have analyzed other hybrid algorithms and found that a combination of PCR and Thomas gave the best overall performance.

The rest of this paper is structured as follows. Section 2 introduces the extended block cyclic reduction algorithm used by the BLKTRI solver. Section 3 gives a brief description of the standard algorithms for solving scalar tridiagonal systems. In Section 4 we detail the mapping of the BLKTRI solver on multicore and GPUs and analyze their performance and then in Section 5 we extend our discussion to 3D problems. Finally, Section 6 concludes summarizing the most relevant contributions.

2. Three Dimensional Elliptic Systems

In this section, we explain the strategy followed to solve a classical 3D Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

defined on a Cartesian domain Ω with prescribed conditions on its boundary $\partial\Omega$.

Discretizing the domain using a Cartesian mesh uniform along each direction, for each (i, j, k) interior node we obtain:

$$\delta_x^2(i, j, k) + \delta_y^2(i, j, k) + \delta_z^2(i, j, k) = f_{i,j,k} \tag{1}$$

where

$$\begin{aligned} \delta_x^2(i, j, k) &= \left(u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}\right)/\Delta x^2 \\ \delta_y^2(i, j, k) &= \left(u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}\right)/\Delta y^2 \\ \delta_z^2(i, j, k) &= \left(u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}\right)/\Delta z^2 \end{aligned}$$

are the finite difference centred approximations to the second derivatives along each direction. The boundary conditions that we will consider are either of Dirichlet or Neumann type on the surfaces normal to the y and z directions and periodic in the x one. The periodic condition aplied in one of the directions allows to uncouple the 3D problem into a set of several independent 2D problems (Figure 1) using a discrete Fourier transform. Hereafter we will briefly explain how the decoupling process takes place. Let N being

the number of equispaced nodes in the x direction that cover the interval $(0, 2\pi)$. We expand the uknown function $u(x, y, z)$ and $f(x, y, z)$ in Fourier series as:

$$u_{n,j,k} = \frac{1}{N}\sum_{l=1}^{N} \hat{u}_{l,j,k} e^{-i\alpha(n-1)} \text{ with } \alpha = \frac{2\pi(l-1)}{N} \tag{2}$$

where $\hat{u}_{l,j,k}$ is the l^{th} Fourier coefficient of the expansion. Next, the expansion is used in equation (1), obtaining the relationship:

$$\frac{1}{N}\sum_{l=1}^{N} e^{-i\alpha(n-1)} \left\{ \frac{\hat{u}_{l,j,k}}{\Delta x^2}(e^{-i\alpha} - 2 + e^{i\alpha}) + \delta_y^2 \hat{u}_{l,j,k} + \delta_z^2 \hat{u}_{l,j,k} \right\} = \frac{1}{N}\sum_{l=1}^{N} \hat{F}_{l,j,k} e^{-i\alpha n} \tag{3}$$

Equation (3) is equivalent to the set of N equations ($l = 1 \cdots N$):

$$\frac{\hat{u}_{l,j,k}(2cos(\alpha) - 2)}{\Delta x^2} + \frac{\hat{u}_{l,j+1,k} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j,k-1}}{\Delta z^2} = \hat{F}_{l,j,k} \tag{4}$$

having used the identity $e^{i\alpha} + e^{-i\alpha} = 2cos(\alpha)$. In short notation (4) reads as:

$$\frac{\hat{u}_{l,j+1,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} + u_{l,j,k-1}}{\Delta z^2} + \beta_l \hat{u}_{l,j,k} = \hat{F}_{l,j,k}, \;\; l = 1 \cdots N \tag{5}$$

with $\beta_l/2 = cos(\alpha) - 1/\Delta x^2 - 1/\Delta y^2 - 1\Delta z^2$. Thus, by considering the Fourier transform (direct FFT) of F one obtains a set of N, 2D independent problems having as unkwowns the Fourier coefficients $\hat{u}_{l,j,k}, l = 1..N$. Each independent problem concerns the solution of a linear system of equations which coefficient matrix is block tridiagonal. Of course, each one of this linear systems can now be solved in a distributed fashion, in parallel. Once the solution is obtained in Fourier space a backward FFT can be used to recast the solution in physical space. Figure (1) provides an algorithmical sketch of the method.

To deal with each decoupled 2D problem, we have chosen a direct method based on a block cyclic reduction algorithm. As shown above, the whole method provides for a blend of coarse and fine-grain parallelism that can be exploited when mapped on heterogeneous platforms.

2.1. Extended Block Cyclic Reduction

In this subsection we briefly summarize a classical direct method for the discrete solution of separable elliptic equations based on a block cyclic reduction algorithm [13]. This method is commonly used when tackling the

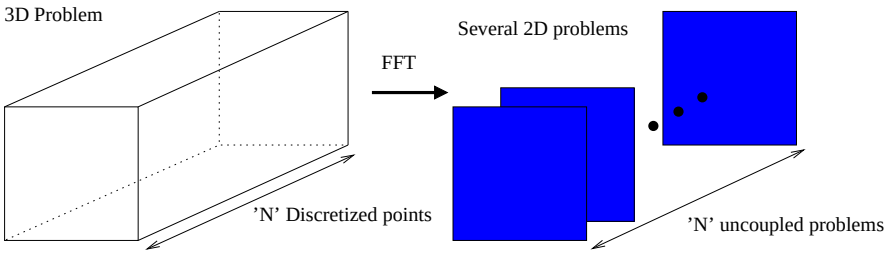

Figure 1: The Fourier based decoupling algorithm

solution of a linear system of equations arising from the second order centered finite difference discretization of 2D separable elliptic equations. From the standpoint of computational complexity (speed and storage), for a $m \times n$ net, its operation count is proportional to $mn \log_2 n$, and the storage requirements are minimal, since the solution is returned in the storage occupied by the right side of the equation (i.e., $m \times n$ locations are required). More in details, consider the 2D separable elliptic equation having $u(x, y)$ as unknown field (Poisson equation is a particular case of what follows):

$$\frac{\partial}{\partial x}\left(a(x)\frac{\partial u}{\partial x}\right) + b(x)\frac{\partial u}{\partial x} + c(x)u + \frac{\partial}{\partial y}\left(d(y)\frac{\partial u}{\partial y}\right) + e(y)\frac{\partial u}{\partial y} + f(y)u = g(x, y) \tag{6}$$

If we discretize (6) with given Dirichlet or Neumann boundary conditions assigned on the edges of a square, using the usual five-point scheme with the discrete variables ordered in a lexicographic fashion, we obtain a linear system of $m \times n$ equations (having m nodes in the x direction and n in y one): $\mathbf{A}\tilde{\mathbf{u}} = \tilde{\mathbf{g}}$, where $\mathbf{A}$ is a block tridiagonal matrix:

$$\mathbf{A} = \begin{bmatrix} B_1 & C_1 & & & & 0 \\ A_2 & B_2 & C_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & A_{n-1} & B_{n-1} & C_{n-1} \\ & & & & A_n & B_n \end{bmatrix}$$

and the vectors $\vec{u}$ and $\vec{g}$ are consistently split as a set of sub-vectors $\vec{u_j}$ and $\vec{g_j}$, $j = 1 \cdots n$, of length m each (i.e., the solution along the j^{th} domain row):

$$\mathbf{u} = [\tilde{\mathbf{u_1}},\ \tilde{\mathbf{u_2}}, \cdots\cdots \tilde{\mathbf{u_n}}]^{\mathbf{T}}$$

$$\mathbf{g} = [\tilde{\mathbf{y_1}},\ \tilde{\mathbf{y_2}}, \cdots\cdots \tilde{\mathbf{y_n}}]^{\mathbf{T}}$$

There is no restriction on m; however, cyclic reduction algorithms require $n = 2^k$, with large values of k for optimal performances. The blocks $\mathbf{A_i}$, $\mathbf{B_i}$ and $\mathbf{C_i}$ are $m \times m$ square matrices. In particular, the *BLKTRI* algorithm requires them to be of the form:

$$A_i = a_i I \tag{7}$$

$$B_i = B + b_i I \tag{8}$$

$$C_i = c_i I \tag{9}$$

where a_i, b_i and c_i are scalars. Having used a standard five point stencil for the discretization of (6), the matrix B is of tridiagonal pattern. The solution is obtained using an *extended cyclic reduction algorithm* which consists of the following phases (more details are found in [13]):

1. **Preprocessing**. This phase consists of computing the roots of certain matrix polynomials. This set of intermediate results only depends on the entries of **A** (not on the right hand side rhs of the equation).
2. **Recursive Reduction**. A sequence of linear systems is generated starting from the original complete one by decoupling odd and even equations. At each step, or level r, about half the unknown vector $\vec{u_i}$ is reduced by eliminating essentially half the remaining unknown vectors until a single unknown vector $\vec{u_2^k}$ remains.
3. **Back-substitution**. The solution vectors $\vec{u_i}$ are determined by first solving the final system generated in the above phase $\vec{u_2^k}$. Then the linear systems are solved in reverse order determining more $\vec{u_i}$ solution vectors, using those $\vec{u_i}$ previously computed.

Overall, during the reduction phase the following equations are solved [13]:

$$q_i^{(r)} = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} p_i^{(r)} \tag{10}$$

$$p_i^{(r+1)} = \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)} + \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)} - p_i^r \tag{11}$$

where $\mathbf{g}$ is split in two different terms, q and p. B stores the roots calculated in the preprocessing phase. This procedure is required to stabilize the method [13]. α and γ have the following form:

$$\alpha_i^{(r)} = \prod_{j=i-2^{r+1}}^{i} aj \tag{12}$$

$$\gamma_i^{(r)} = \prod_{j=1}^{i+2^{r-1}} cj \tag{13}$$

Conversely, in the last phase the following equations are solved [13]:

for $r = k, k-1, \ldots, 0$ and $i = 2^r, 3 \times 2^r, 5 \times 2^r, \ldots, 2^{k-r} \times 2^r$:

$$u_i = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} \left[p_i^{(r)} - \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} u_{i-2^r} - \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} u_{i+2^r} \right] \tag{14}$$

This method is implemented in a FORTRAN package library called FISHPACK (the *BLKTRI* routine), which is widely used and well known within the computational fluid dynamics community[14].

To obtain the aforementioned terms the solution of a set of scalar tridiagonal systems of equations must be faced. The solution of these systems represent the most expensive stage of the algorithm. Nevertheless, other basic mathematical operations such as vectors sums or scalar vector multiplications introduce a non negligible cost.

3. Parallel Tridiagonal Algorithms

As mention above, a key element of the BLKTRI algorithm is how to solve a set of scalar tridiagonal systems. The original BLKTRI implementation in the *fishpack* package makes use of the Thomas algorithm (TA) [15]. TA is a specialized application of the Gaussian elimination that takes into account the tridiagonal structure of the system. TA consists of two stages, commonly denoted as forward elimination and backward substitution.

Given a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

The forward stage eliminates the lower diagonal as follows:

$$c'_1{=}\frac{c_1}{b_1}, \quad c'_i{=}\frac{c_i}{b_i{-}c'_{i-1}a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$
$$y'_1{=}\frac{y_1}{b_1}, \quad y'_i{=}\frac{y_i{-}y'_{i-1}a_i}{b_i{-}c'_{i-1}a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$

and then the backward stage recursively solve each row in reverse order:

$$u_n = y'_n, u_i = y'_i - c'_i u_{i+1} \text{ for } i = n-1, n-2, \ldots, 1$$

Overall, the complexity of TA is optimal: $8n$ operations in $2n-1$ steps. Unfortunately, this algorithm is purely sequential.

Cyclic Reduction (CR) [21, 7, 4] is a parallel alternative to TA. It also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_i x_{i-1}{+}b_i x_i{+}c_i x_{i+1} = d_i$ are reduced. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2 \; c'_i = -c_{i+1}k_2, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2$$
$$k_1{=}\frac{a_i}{b_{i-1}}, k_2{=}\frac{c_i}{b_{i+1}}$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns x_i are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i{=}\frac{y'_i{-}a'_i x_{i-1}{-}c'_i x_{i+1}}{b'_i}$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps. Figure 2 graphically illustrates its access pattern.

Parallel Cyclic Reduction (PCR) [22, 7, 4] is a variant of CR, which only has substitution phase. For convenience, we consider cases where $n = 2^s$, that involve $s = \log_2 n$ steps. Similarly to CR a, b, c and y are updated as follows, for $j = 1, 2, \ldots, s$ and $k = 2^{j-1}$:

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$
$$\alpha_i{=}\frac{-a_i}{b_{i-1}}, \beta_i{=}\frac{-c_i}{b_i}$$

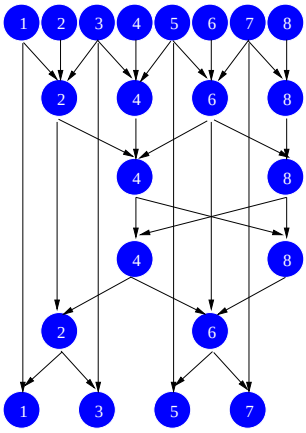

Figure 2: Access pattern of the CR algorithm.

finally the solution is achieved as:

$$u_i = \frac{y'_i}{b_i}$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations. Overall, the operation count of PCR is $12n \log_2 n$. Figure 3 sketches the corresponding access pattern.

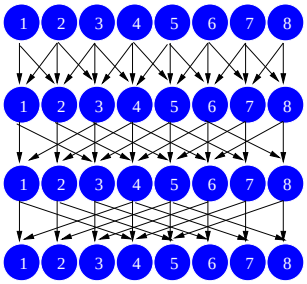

Figure 3: Access pattern of the PCR algorithm.

We should highlight that apart from their computational complexity these algorithms differ in their data access and synchronization patterns, which also have a strong influence on their actual performance. For instance, in the CR algorithm synchronizations are introduced at the end of each step and its corresponding memory access pattern may cause bank conflicts. PCR needs less

steps and its memory access pattern is more regular [7]. In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [7, 1, 2, 4]. Figure 4 illustrates the access pattern of the CR-PCR combination proposed in [7]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then solves the reduced (intermediate) system with the PCR algorithm. Finally, it substitutes the solved unknowns back into the original system using the backward substitution phase of CR.

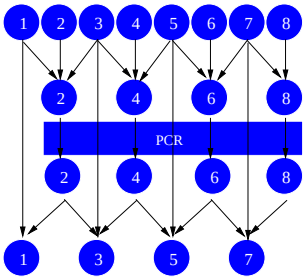

Figure 4: Communications pattern for the CR-PCR algorithm.

4. Parallel Block Cyclic Reduction

The reduction and back-substitution phases are the core of the BLKTRI algorithm. In this section we focus on describing how to map these phases onto GPUs. The mapping on multicore systems requires less transformations to the original BLKTRI code. In this case, the most effective scheme consists in using a coarse-grain strategy for distributing the independent tridiagonal problems that arise at the different steps of the algorithm across the different cores. This way, these tridiagonal systems are solved sequentially on each core using the optimal TA algorithm. This distribution is well balanced and data locality is optimized mapping a subset of continuous systems onto each core. The original FISHPACK BLKTRI routine can be easily parallelized with this approach annotating some of its loops with Open-MP pragmas.

For our mapping on GPUs, we have identified four main kernels, which are graphically illustrated, along with their dependencies, in Figure 5. All data need to be uploaded to the GPU memory before launching the q kernel and finally, the solution u is transferred back to the CPU memory. For convenience, we have denoted $\alpha_i^r(B_{i-2^{r-1}}^{r-1})^{-1}q_{i-2^r}^{(r)}$ as α and $\gamma_i^r(B_{i+2^{r-1}}^{r-1})^{-1}q_{i+2^r}^{(r)}$ as γ. The p kernel consists on the addition of three vectors. The core of

the computation is performed in the remaining three kernels, which share a similar pattern sketched in the *Generic Tridiagonal Kernel*. Essentially, these three kernels solve an independent set of tridiagonal systems but differ on their pre- and post-processing calculations.

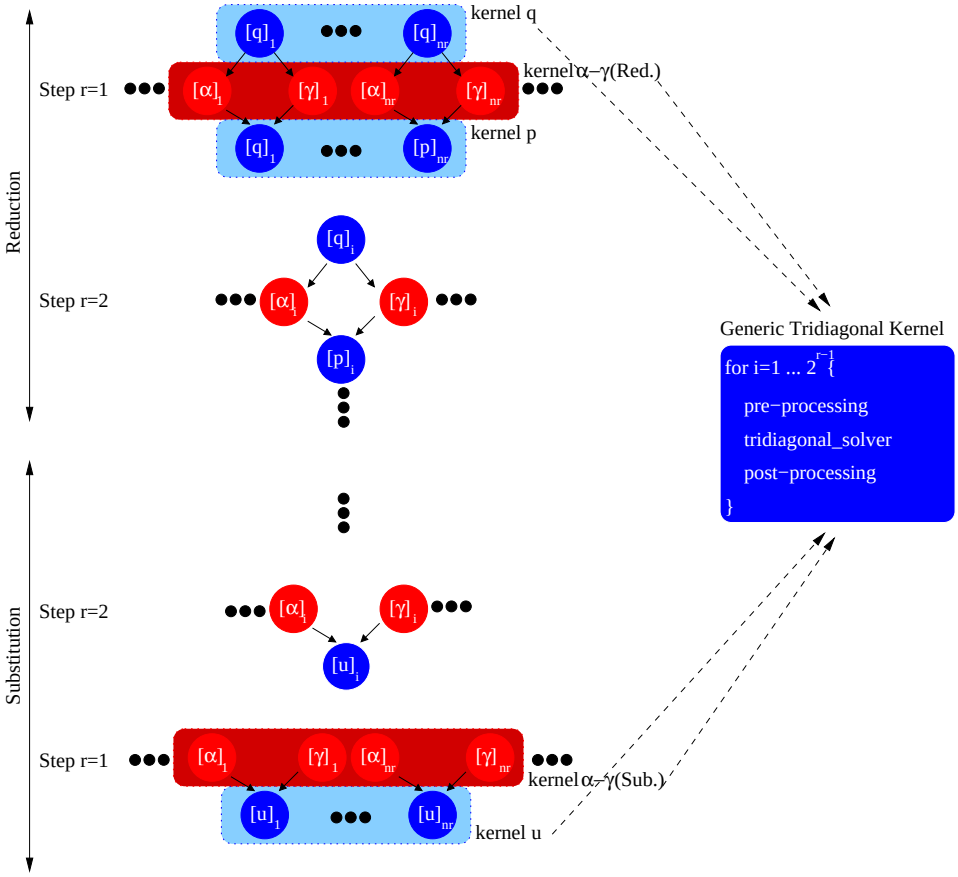

Figure 5: Main kernels of the reduction and substitution phases of the BLKTRI algorithm.

Figure 6 illustrates with more detail the mapping of the generic kernel on the GPU. Figure 6-top shows a coarse-grain scheme similar to the multi-core counterpart. In this coarse distribution a set of tridiagonal systems is mapped onto a CUDA block so that each CUDA thread fully solves a system using the TA algorithm. Unfortunately, this approach, which is relatively easy to implement, does not exploit efficiently the memory hierarchy of the GPU since the memory footprint of each CUDA thread becomes too large. Previous research has shown that fine-grain alternatives based on PCR are

more efficient [7, 4, 9]. In this case (Figure 6-bottom), each tridiagonal system is distributed across the threads of a CUDA block so that the shared memory of the GPU can be used more effectively (both the matrix coefficients and the right hand side of each tridiagonal system are hold on the shared memory of the GPU).

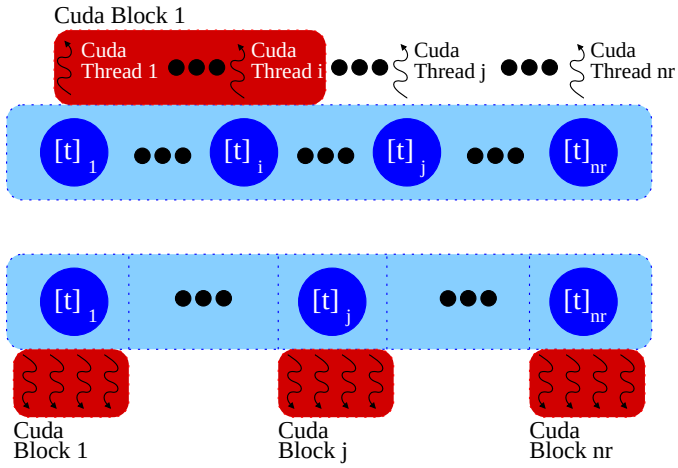

Figure 6: Coarse (top) and fine (bottom) distributions of the generic kernel.

We should highlight that the arithmetic intensity of our generic tridiagonal kernels is higher than the synthetic tridiagonal benchmarks analyzed by previous research [7, 4, 5]. This is an advantage when using the GPU as an accelerator since the impact of CPU-GPU data transfers on performance is much lower.

Figure 7-top (strong scaling) compares the different approaches using as a simplified test a single 1024×1024 2D problem. This is a relatively small 2D problem but note that it arises from the solution of a 3D problem. These tests have been run on an heterogeneous platform, whose main features are 2 CPUs Intel E5-2650 (up to 8 cores and 16 threads per processor) with 128 GB DDR3-1600 of RAM memory and 1 GPU nVidia K20c (Kepler) with 2496 CUDA cores and 5 GB GDDR5 of device memory. We have used Fedora Linux 16 and the compiler The Portland Group (PGI) Fortran (flags -fast -Mipa=fast,inline -mp -Mcuda). Each step of the algorithm has a different level of parallelism but, as shown in Figure 7-top, the computational load of all of them are similar since as the level of parallelism reduces, the number of *iterations* of the generic tridiagonal kernel increases. The parallel

implementations outperform the sequential BLKTRI routine in most cases. Only for the steps with reduced parallelism, the GPU version becomes ineffective. This is the expected behavior since in these cases the number of CUDA blocks is very small, being just one CUDA block in the last (first) step of the reduction (substitution) phase.

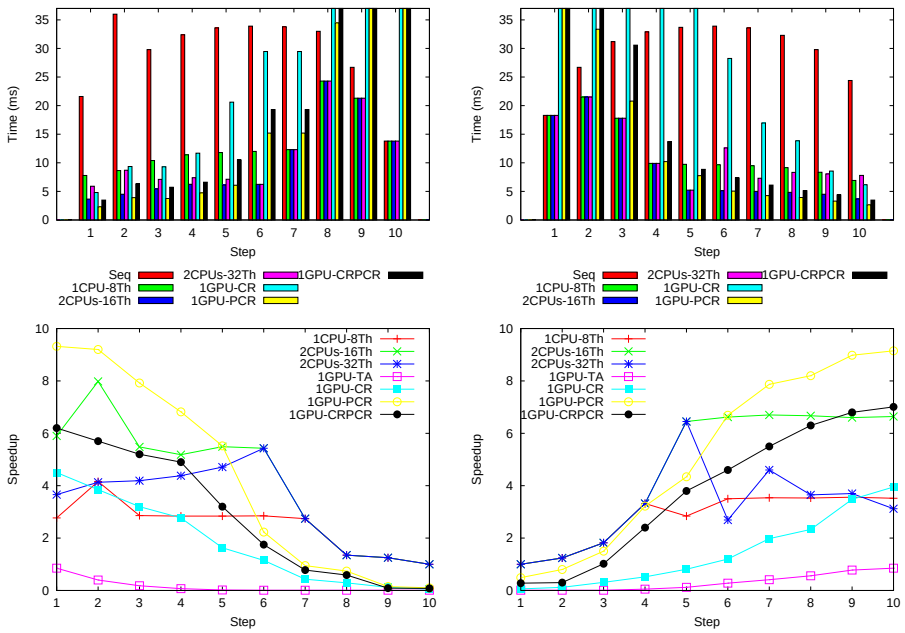

Figure 7: Execution time (top) and Speedup (bottom) on each step of the reduction (left) and substitution (right) phases of the extended 2D block cyclic reduction.

Figure 7-bottom shows the speedup at each step of the extended block cyclic reduction algorithm over the sequential counterpart. As mention above, the coarse thread distribution based on the TA algorithm does not perform well on GPUs, but on multicore, this coarse approach does provide satisfactory speedups across all steps despite this is a small problem, achieving best performance when running 16 threads on 2 CPUs and 16 cores. On GPUs, PCR provides satisfactory speedups on the first (last) steps of the reduction (substitution) phases and is able to outperform CR and the hybrid CR-PCR algorithms across all steps.

According to these results, the optimal approach appears to be an heterogeneous combination of PCR on the GPU and TA (16 threads) on multicore for those steps with lower parallelism. Nevertheless, this combination

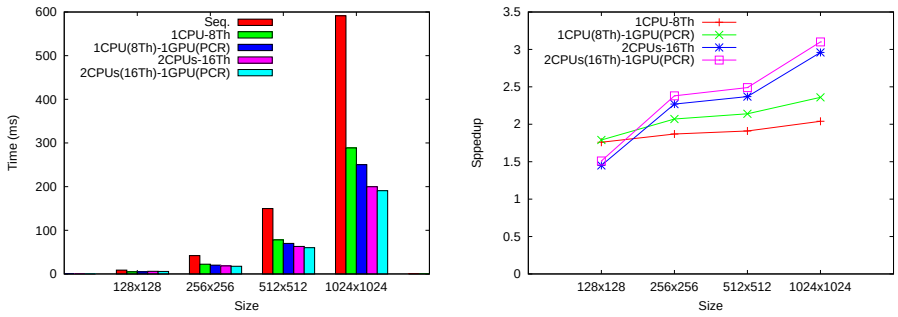

Figure 8: Total execution time (left) and trend of the speedup (right) increasing the size of the problem.

requires additional CPU-GPU data transfers that may degrade the actual performance. Figure 8 shows the overall speedups on Kepler for different problem sizes taking into account these data transfer overheads. We have fixed 5 different computing platforms and increased the size of the problem to carry out a weak scaling study. The heterogeneous approach is able to outperform the homogeneous multicore counterpart in all cases. A fully homogeneous GPU implementation (not shown in Figure 8) does not provide satisfactory results since in those steps with low parallelism, its performance becomes very inefficient.

5. Parallel Three Dimensional Elliptic Systems

In this Section we present the proposed approaches to solve in parallel a Three Dimensional Elliptic Systems problem on heterogeneous platforms. The FFT method can be computed in parallel on both multicore and GPUs platforms. This is a well know problem and there are several libraries that provide satisfactory results [18, 19, 20]. We have focused instead on solving the set of independent 2D problems in parallel, which is the main contribution of this work. In the 2D case, the homogeneous GPU implementation does not provide satisfactory results and the heterogeneous counterpart is able to achieve the best performance. We want to know if this is still valid for the 3D problem.

Figure 9-left graphically sketches the parallel profile of a 2D problem. We have highlighted three different stages: two of them have a high level of parallelism (blue areas) while the red one has limited parallelism as explained in the previous Section. In the 3D case (Figure 9-right) we need to solve a set

of independent 2D problems and hence, the amount of parallelism increases in all the steps by a *problem size* factor, which we have denoted as the S factor.

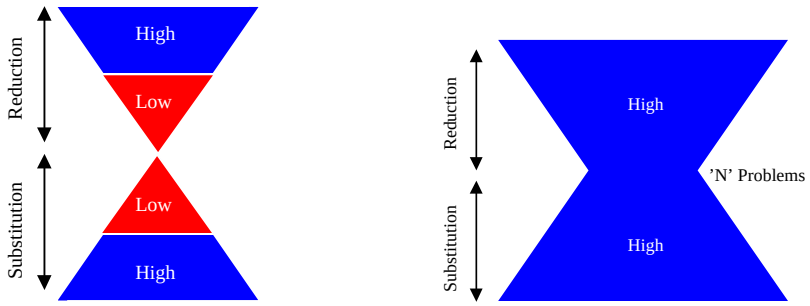

Figure 9: Amount of parallelism for one 2D block tridiagonal problem (left) and for a set of independent 2D block tridiagonal problems (right).

The mappings of a 3D problem on our target computing platforms are similar to their 2D counterparts. On multicore, we follow a coarse-grain approach mapping a set of 2D problems on each core, which are solved sequentially using the TA algorithm. On GPUs, we follow a fine-grain approach based on the PCR algorithm, which is essentially the same as the 2D case. The major different is the number of CUDA blocks at each step, which is S times higher in 3D. An heterogeneous combination of the multicore and GPU implementation is also possible, as in the 2D case. The data transfers overheads are potentially much lower than in the 2D case since it is possible to perform them asynchronously. As shown graphically in Figure 10, this allows the overlapping of data transfers with useful computation on the GPUs or the CPUs, and indirectly, of GPUs with CPUs computation.

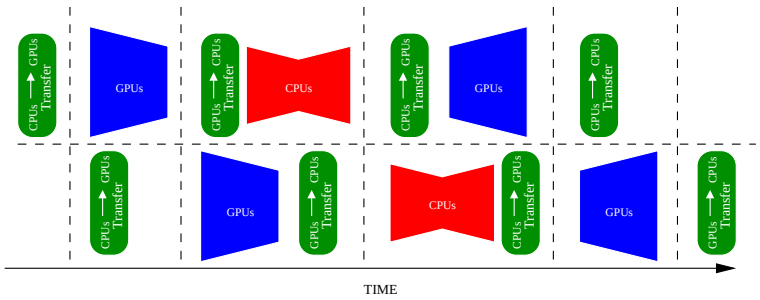

Figure 10: Heterogeneous approach.

Figure 11 (strong scaling) compares the homogeneous multicore and GPU implementations using as a test case a $512 \times 512 \times 512$ problem. Unlike the 2D case, the speedup figures are less dependent on the step of the algorithm due to the higher level of parallelism. In fact, in all the steps the speedup figures are close to the highest speedup attainable in the 2D case. Another important consequence is that the GPU version always outperforms the multicore counterpart.

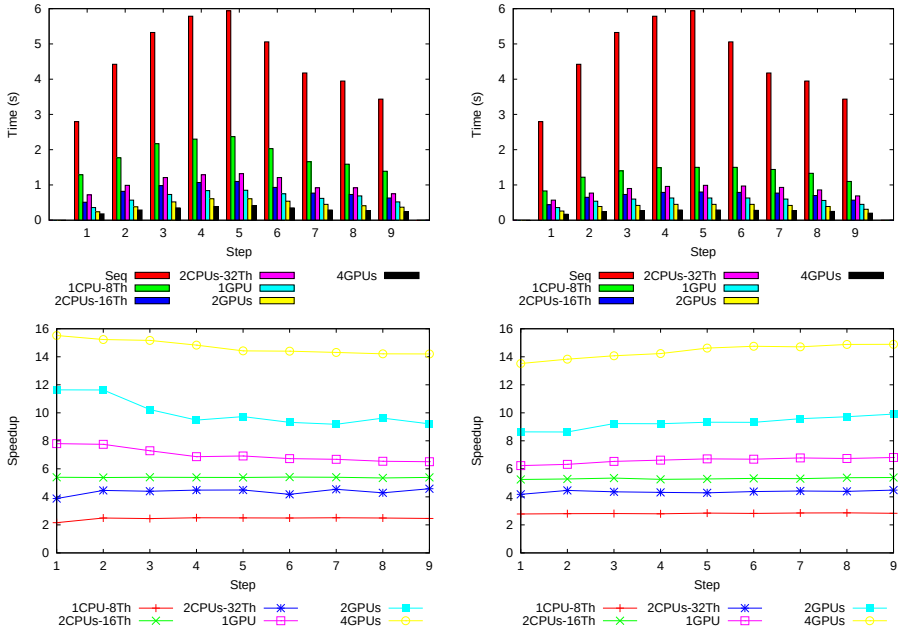

Figure 11: Execution time (top) and speedup (bottom) obtained in each step in the reduction (left) and substitution (right) phases.

These results question the potential benefits that the heterogeneous strategy could achieve since it seems that the homogeneous GPU implementation is able to exploit all the available parallelism. Figure 12 (weak scaling) compares the homogeneous and heterogeneous approaches. First, we should highlight that in the homogeneous GPU implementation data transfers incur a very small overhead (lower than 2% of the execution time). This is the expected behavior since the arithmetic intensity of the 3D problem is very high. In spite of this, the heterogeneous approach is able to outperform the homogeneous counterpart since it benefits from actual GPUs-CPUs overlapping. As shown in Figure 12, these gains grow with the "S" factor. Overall,

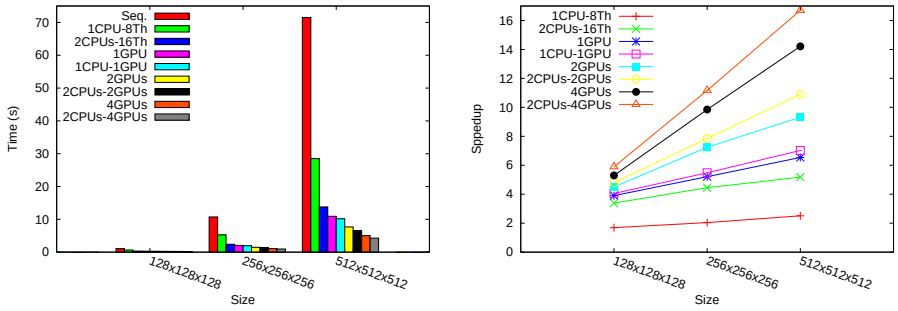

Figure 12: Execution time (s) (left) and speed up (right) for all the implementations.

the extra benefit of the heterogeneous implementation can reach up to 15 % in terms of execution time over the homogeneous GPU counterpart.

6. Conclusions

The efficient solution of block tridiagonal linear systems is of crucial importance since it is the major bottleneck of several large scale simulation codes dealing with time-dependent elliptic partial differential equations. We have analyzed the performance of different parallel implementation that exploit homogeneous multicores and GPUs systems, as well as heterogeneous multicores-GPUs platforms.

On multicore, a coarse grain approach based on the Thomas algorithm is the best option for solving the intermediate scalar tridiagonal systems that arise on both 2D and 3D problems. In contracts, on GPUs it is much better a fine grain alternative based on using the PCR algorithm.

As expected, an heterogeneous approach that combines both implementations is the best option on 2D problems. For 3D problems, we have shown that this is also the best choice despite the 3D problem has both higher arithmetic intensity and higher parallelism than the 2D case. Indeed, the homogeneous GPUs implementation outperform the multicore counterpart even for medium size 3D problems. However, the heterogeneous approach benefits from CPUs-GPUs overlapping an is able to achieve an additional 15% performance gain.

Acknowledgments

This work has been supported by the Spanish Consolider grant Supercomputación y e-Ciencia (SyeC) (Ref: CSD2007-00050).

References

[1] N. Sakharnykh, Efficient tridiagonal solvers for ADI methods and fluid simulation, NVIDIA GPU Technology Conference, September 2010.

[2] A. Davidson, Y. Zhang, and J. D. Owens, An auto-tuned method for solving large tridiagonal sysstonetems on the GPU, in Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, May 2011.

[3] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, R. Sanchez, BCYCLIC: A parallel block tridiagonal matrix cyclic solver, Journal of Computational Physics, Volume 229, Issue 18, 2010, 6392-6404.

[4] H.-S. Kim, S. Wu, L.-W. Chang, W.-m. W. Hwu: A Scalable Tridiagonal Solver for GPUs. ICPP 2011: 444-453.

[5] C. P. Stone, E. P. N. Duque, Y. Zhang, D. Car, J. D. Owens, and R. L. Davis. GPGPU parallel algorithms for structured-grid CFD codes. In Proceedings of the 20th AIAA Computational Fluid Dynamics Conference, number 2011-3221, June 2011.

[6] D. Geer. Chip markets turn to multicore processors. Computer, 38(5):11-13, 2005.

[7] Y. Zhang, J. Cohen, J. D. Owens. Fast Tridiagonal Solvers on the GPU. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 127-136, 2010.

[8] D. Goddeke, R. Strzodka. Cyclic Reduction Tridiagonal Solvers on GPU Applied to Mixed-Precision Multigrid. IEEE Transactions on Parallel and Distributed System, 22:22-32, 2010.

[9] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, Manuel Prieto-Matías. Block Tridiagonal Solvers on Heterogeneous Architectures. The 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012.

[10] W.-C. Feng, D. Manocha. High-performance computing using accelerators, Parallel Computing, Elsevier, 33:645-647, 2007.

[11] GPGPU. *General-purpose computation using graphics hardware*. http://www.gpgpu.org.

[12] J. L. Guermond, P. Minev, J. Shen. An overview of projection methods for incompressible flows. Comput. Methods Appl. Mech. Engrg. 195(44-47): 6011-6045, 2006.

[13] P. N. Swarztrauber. A Direct Method for the Discrete Solution of Separable Elliptic Equations. SIAM J. Numer. Anal. 11:1136-1150, 1974.

[14] FISHPACK. http://www.cisl.ucar.edu/css/software/fishpack/

[15] S.D. Conte, C. de Boor. Elementary Numerical Analysis, McGraw-Hill, 1976.

[16] R. Buchty, V. Heuveline, W. K., J.-P. Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. Concurrency and Computation: Practice and Experience 24(7): 663-675, 2012.

[17] J. Nickolls, W. J. Dally. The GPU Computing Era. IEEE Micro 30(2): 56-69, 2010.
Parallel Computing, Elsevier, 33:645-647, 2007.
Cg: a system for programming graphics hardware in a C-like language. SIGGRAPH'03: ACM SIGGRAPH, 896-907, 2003.

[18] FFT OpenMP, Fast Fourier Transform Using OpenMP. http://people.sc.fsu.edu/~jburkardt/c_src/fft_openmp/fft_openmp.html

[19] 3D FFT. http://charm.cs.uiuc.edu/cs498lvk/projects/kunzman/index.htm

[20] NVIDIA. The NVIDIA CUDA Fast Fourier Transform library (cuFFT). http://developer.nvidia.com/cuda/cufft

[21] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. Journal of the ACM, 12(1):95-113, 1965.

[22] R. W. Hockney and C. R. Jesshope. Parallel Computers. Adam Hilger, Bristol, 1981.